IENAC-S-95 Octobre 1996

Corrigé de logique et calculabilité

1 Logique

1. Nommons la proposition «un algorithme est simple» par s et «un algorithme est correct» par c.
   - Si le «ou» est classique, le jugement s'écrit $(s \rightarrow c) \rightarrow (c \vee \neg s)$ équivalent à $(c \vee \neg s) \rightarrow (c \vee \neg s)$. Or $F \rightarrow F$ est une tautologie (table de vérité) donc le jugement également.
   - Si le «ou» est exclusif, le jugement s'écrit $(s \rightarrow c) \rightarrow (s \wedge \neg c \vee c \wedge \neg s)$. Cette formule est fausse pour s et c faux, donc ça n'est pas une tautologie.
2. La formalisation peut être faite en calcul des prédicat. Soient les symboles de prédicat suivant :
   - $dans(x, y)$ pour «x est dans la pièce y» ;
   - $voisin(x, y)$ pour «x est une pièce voisine de y» ;
   - $audible(x)$ pour «le coup de feu était audible depuis la pièce x» ;
   - $entend(x)$ pour «x a entendu le coup de feu» ;

   et les symboles de constante suivant ;
   - c pour «la cuisine» ;
   - s pour «la salle à manger» ;
   - m pour «le majordome».

   Les faits s'expriment alors par l'ensemble de formules suivant (en supposant que la bonne et le majordome disent la vérité) :

   $$dans(m, s) \tag{1}$$
   $$voisin(c, s) \tag{2}$$
   $$\forall x \; voisin(c, x) \rightarrow audible(x) \tag{3}$$
   $$\forall x \forall y \; (audible(x) \wedge dans(y, x)) \rightarrow entend(y) \tag{4}$$
   $$\neg entend(m) \tag{5}$$

   On montre alors par résolution que cet ensemble de formules (ce sont des clauses) n'est pas satisfiable :

   $$(5) \text{ et } (4) \vdash \neg audible(x) \vee \neg dans(m, x) \tag{6}$$
   $$(6) \text{ et } (1) \vdash \neg audible(s) \tag{7}$$
   $$(7) \text{ et } (3) \vdash \neg voisin(c, s) \tag{8}$$
   $$(8) \text{ et } (2) \vdash \square$$

3.

   $$l(n) \tag{9}$$
   $$\forall x \forall y \; l(y) \rightarrow l(c(x, y)) \tag{10}$$
   $$\forall x \forall y \; mb(x, c(x, y)) \tag{11}$$
   $$\forall x \forall y \forall z \; mb(x, y) \rightarrow mb(x, c(z, y)) \tag{12}$$
   $$\forall y \; ap(n, y, y) \tag{13}$$
   $$\forall e \forall x \forall y \forall z \; ap(x, y, z) \rightarrow ap(c(e, x), y, c(e, z)) \tag{14}$$

   On remarque que toutes ces formules sont des clauses.

(a) Résolution signifie réfutation : on calcule la négation de la formule à prouver et on la skolémise :

$$\neg mb(a, c(b, c(a, n))) \tag{15}$$

avec a et b deux nouvelles constantes d'arité 0. On tente ensuite d'obtenir la clause vide par résolution :

$$\begin{array}{rcl} (15) \text{ et } (12) & \vdash & \neg mb(a, c(a, n)) \quad (16) \\ (16) \text{ et } (11) & \vdash & \Box \end{array}$$

(b) On remarque que seule les formules 13 et 14 sont susceptibles d'être utiles à la preuve. D'où l'arbre de preuve suivant :

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\Sigma + a + b; (13), (14), \underline{ap(n, t_5, t_5)} \longrightarrow \underline{ap(n, b, b)}}\ \text{axiome avec } t_5 = b \text{ un } \Sigma + a + b\text{-terme}}{\Sigma + a + b; \underline{(13)}, (14) \longrightarrow ap(n, b, b)}\forall_G^1}{\Sigma + a + b; (13), (14), \underline{ap(t_1, t_2, t_3) \to ap(c(t_4, t_1), t_2, c(t_4, t_3))} \longrightarrow ap(c(a, n), b, c(a, b))}\to'_G \text{ avec } \dagger}{\Sigma + a + b; (13), \underline{(14)} \longrightarrow ap(c(a, n), b, c(a, b))}\forall_G^4}{\Sigma; (13), (14) \longrightarrow \underline{\forall a \forall x\ ap(c(a, n), x, c(a, x))}}\forall_D^2}{\Sigma; \longrightarrow \underline{(13) \wedge (14) \to \forall a \forall x\ ap(c(a, n), x, c(a, x))}}\to_D \wedge_G$$

$\dagger$: $t_4 = a$, $t_1 = n$, $t_2 = b$ et $t_3 = b$ des $\Sigma + a + b$-termes.

(c) i. Les éléments de $\mathcal{L}$ sont les listes d'entiers.

ii. – $\hat{n} =<>$ la liste vide («nil»)
– $\hat{c}(x, y) =< x, y >$ le constructeur de liste («cons»)
– $\hat{l}(x)$ exprime $x \in \mathcal{L}$ («list»)
– $\widehat{mb}(x, y)$ exprime x est un élément de la liste y («member»)
– $\widehat{ap}(x, y, z)$ exprime z est la concaténation de x et y («append»)

2 Calculabilité

1. `z := x; do y times z := z + 1 end`
2. Si $\mathcal{L}$ était TURING-complet, le problème de l'arrêt pour les programmes de $\mathcal{L}$ serait indécidable. Or tous les programmes de $\mathcal{L}$ terminent : les boucles sont bornées. Donc $\mathcal{L}$ n'est pas TURING-complet.

   Malgré les apparences, beaucoup de calculs sont exprimables en $\mathcal{L}$. Pour calculer le prédécesseur de x, on peut faire :

   ```
   a := 0; b := 0; do x times b := a; a := a + 1 end; x := b
   ```

   D'où pour calculer $x - y$:

   ```
   do y times a := 0; b := 0; do x times b := a; a := a + 1 end; x := b end
   ```

   Une conditionnelle `if c then` t `else` e s'écrira (en codant vrai avec 1 et faux avec 0) :

   ```
   nc := calcul de (1-c)
   do c times t end
   do nc times e end
   ```

   Pour représenter les nombre flottants, c'est moins simple ... mais pas plus compliqué qu'avec une machine de TURING.

3. En fait, on remarque que le langage $\mathcal{L}$ est équivalent à la classe des fonctions primitives récursives. Pour obtenir la TURING-complétude, il suffit de rajouter une construction analogue à la minimisation non bornée (schéma μ des fonctions récursives). La construction `while` fait l'affaire :

$$programme \quad ::= \quad \texttt{while}\, variable\, \texttt{do}\, programme\, \texttt{end}$$

4. Avec une implémentation sur des mots de 32 bits du langage $\mathcal{L}'$, le nombre de configurations, c-à-d le contenu des variables (qui sont en nombre fini), est fini donc le problème de l'arrêt est décidable : lors de l'exécution, si une configuration précédente est retrouvée, on peut conclure que le programme boucle.